Polaris: Enhancing CXL-based Memory Expanders with Memory-side Prefetching

Zhe Zhou[1,2], Shuotao Xu[4], Yiqi Chen[1], Tao Zhang[4], Ran Shu[4], Lei Qu[4], Peng Cheng[4], Yongqiang Xiong[4], and Guangyu Sun[1,2,3(✉)]

[1] School of Integrated Circuits, Beijing, China
{pkuzhou,cyq1009,gsun}@pku.edu.cn
[2] School of Computer Science, Beijing, China
[3] Beijing Advanced Innovation Center for Integrated Circuits, Beijing, China
[4] Microsoft Research Asia, Beijing, China
{shuotaoxu,zhangt,ran.shu,lei.qu,pengc,yongqiang.xiong}@microsoft.com

Abstract. The use of CXL-based memory expanders introduces increased latency compared to local memory due to control and transmission overheads. This latency difference negatively impacts tasks that are sensitive to latency. While cache prefetching has traditionally been used to mitigate memory latency, addressing this performance gap requires improved CPU prefetch coverage. However, tuning a CPU prefetcher for CXL memory necessitates costly CPU modifications and can result in cache pollution and wasted memory bandwidth. To address these challenges, we propose a solution called Polaris, a novel CXL memory expander that integrates a hardware prefetcher in the CXL memory controller chip. Polaris analyzes incoming memory requests and prefetches cachelines to a dedicated SRAM buffer without requiring modifications to CPUs or software. In cases where prefetch hits occur, Polaris establishes a "shortcut" for rapid memory access, significantly reducing the performance gap between CXL and local DDR memory. Furthermore, if small CPU changes are allowed, such as extending Intel's DDIO, Polaris can further minimize CXL memory access overheads by actively pushing high-confidence prefetches to the CPU's last-level cache (LLC). Extensive experiments demonstrate that, in conjunction with various CPU-side prefetchers, Polaris enables up to 85% of common workloads (on average, 43%) to effectively tolerate CXL memory's longer latency.

Keywords: CXL · Cache Prefetching · Near-memory processing

1 Introduction

Recently, Compute Express Link (CXL) interconnected memory expanders (CXL memory) have been proposed as a new expansion approach to scale up a single server's memory capacity and bandwidth [19,23,42]. Unlike previous methods such as memory expansion through PCIe [29] or RDMA over

C. Li et al. (Eds.): APPT 2023, LNCS 14103, pp. 19–39, 2024.
https://doi.org/10.1007/978-981-99-7872-4_2

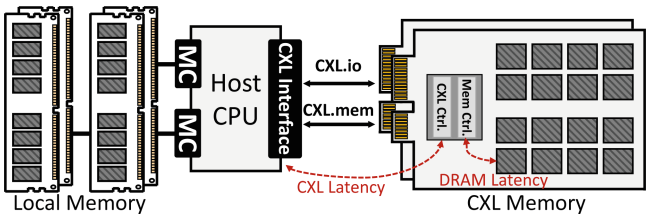

Fig. 1. A server system with both local DDR memory and CXL memory.

Table 1. Feature comparison across different memory types

Type	Interconnect	Latency	Bandwidth	Access Semantic
Local Memory	DDR Channel	80-140 ns	38.4 GB/s[#]	Load/Store
Memory Blade	PCIe	450 ns [29]	64 GB/s*	DMA
RDMA [20]	Infiniband	>1 µs	6 GB/s	DMA
CXL Memory	**CXL Channel**	**170–250 ns** [30]	**64 GB/s**	**Load/Store**

[#]DDR5-4800, single channel. *Scaled to DDR5 & PCIe 5.0 x16.

InfiniBand/Ethernet networks [3–5,18,20,37,40,45] , CXL memory is byte-addressable via normal CPU load/store instructions and exposes a coherent, unified memory space with the local memory. Because a host accesses CXL memory directly without invoking page faults or DMA operations, CXL memory achieves much lower latency than the RDMA and Memory Blade [29] counterparts. This sheds new light on memory expansion in data centers.

However, due to non-negligible control and transmission overheads of the CXL interconnect (the *CXL Latency* in Fig. 1), CXL memory still has ∼2× access latency than local memory accesses, as shown in Table 1. Consequently, many latency-sensitive workloads even suffer up to 50% slowdown on CXL memory [28]. Considering that the CPU accesses CXL memory through normal `load/store` interfaces at a cacheline granularity, one strawman solution to mitigate such a performance gap is to adopt cache prefetching. Theoretically, if most of cache misses are covered by prefetching, average access latency to both local and CXL memory is reduced substantially, and so is the performance gap. However, our profiling reveals that even the state-of-the-art CPU prefetchers cannot provide sufficiently high prefetch coverage to achieve this goal for CXL memories. For a certain CPU prefetcher to hide CXL memory latency, it often requires expanding its prefetch coverage via more aggressive prefetching, and is usually at the cost of lower prefetch accuracy [9]. More aggressive CPU-side prefetching often results in extraneous DRAM accesses that lead to ***cache pollution*** and ***bandwidth waste*** issues. This paradox between prefetch coverage and accuracy makes it challenging to improve CPU prefetchers' performance further for CXL memories. Moreover, incorporating a more accurate prefetcher for CXL memory incurs ***costly modifications to CPUs*** and may ***demand much more core-side resources***. All these limitations indicate that one shall

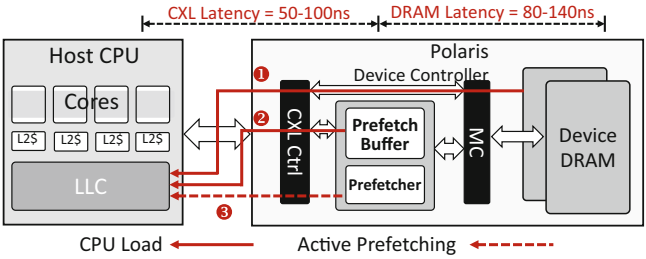

Fig. 2. Illustration of Polaris. Path ❶: CPU loads from device DRAMs. Path ❷: CPU loads from the prefetch buffer. Path ❸: The memory-side prefetcher directly pushes data to CPU's LLC via DDIO.

go beyond CPU side and seek new prefetch opportunities on the CXL side to hide long memory access latency.

In this paper, we propose Polaris, a novel CXL memory expander that reduces the CXL memory latency by ***prefetching on the memory side***. The overall architecture of Polaris is illustrated in Fig. 2. A standard CXL memory expander only has Path ❶, where CPU accesses would suffer from CXL latency and DRAM latency. Polaris creates fast paths for CPU accesses (Paths ❷ and ❸) by adding prefetch functionality to CXL memory. Polaris incorporates a hardware prefetcher in its controller chip, which predicts CPU cacheline accesses and prefetches them to a dedicated SRAM buffer (Prefetch Buffer) for quick future accesses. In events of prefetch hits, Polaris redirects CPU requests to a ***shortcut*** (Path ❷) to substantially reduce the access latency.

The base design of Polaris (Path ❷) already brings several advantages: (1) Hardware modifications are restricted to memory expanders, which facilitates a drop-in compatible solution to existing data-center servers. (2) With a dedicated prefetch buffer, Polaris can unlock more prefetch coverage than CPU prefetchers by aggressive prefetching without polluting the CPU cache. (3) Polaris can harvest the higher device-side DRAM bandwidth than CPU-side for prefetching. (4) Memory-side prefetchers in standalone off-chip silicons have more budget for sophisticated prefetchers than CPU-side prefetchers to yield higher prefetch accuracy. In particular, Polaris ensembles multiple prefetchers and proposes a score-based selector to choose the best-performing prefetcher dynamically. Besides Polaris-Base, we further present Polaris-Active to make the most of memory-side prefetching capability. It actively pushes prefetched cachelines to CPU's LLC to reduce prefetch hit latency (Path ❸) further. We propose that Polaris-Active only requires minimal modifications to the existing direct-cache-access interfaces like Intel's DDIO (Data-Direct-IO) [24]. Extensive experiments on 33 representative workloads demonstrate that together with dif-

ferent CPU prefetchers, POLARIS helps up to 85% of workloads, 43% on average, effectively tolerate[1] CXL memory's longer latency (Sect. 4).

2 Background and Motivation

2.1 CXL-Based Memory Expansion

The emerging Compute Express Link protocol (CXL) [17] is the first open industry standard to support cache-coherent interconnect between the host CPU and various accelerators or memory devices. It is composed of three sub-protocols: `CXL.io` creates high-speed I/O channels called *FlexBus* based on the PCIe-5.0 physical layer. It provides a basic, non-coherent load/store interface for general I/O devices. `CXL.cache` further adds cache coherence abilities to *FlexBus*, which works on MESI coherence protocol and enables the CXL devices to cache the host memory. The third one, `CXL.mem`, allows the host to have coherent, byte-addressable access to the device-attached memory.

The CXL-based memory expanders (*CXL memory*) are built upon `CXL.io` and `CXL.mem`. As shown in Fig. 1, in a system that equips CXL memory, the local and CXL memory have a unified physical memory space. LLC (Last-Level-Cache) misses to CXL memory addresses are translated into CXL requests and sent via CXL channels. At the CXL memory side, these requests will first be decoded by a CXL controller and then fed into the memory controller to access device DRAMs. Responses carrying missed cachelines are sent back to the CPU without invoking page faults or DMAs. Recently, Samsung [42] and SK Hynix [23] have launched commodity CXL memory expanders. They can extend the single server memory capacity to several TBs and provide hundreds of GB/s of extra memory bandwidth. Gouk et al. also implemented an FPGA prototype [19] to demonstrate CXL memory's unmatched advantages over RDMA-based solutions.

2.2 The Long Latency Issue of CXL Memory

As compared in Table 1, though CXL memory has much lower latency than the RDMA/PCIe-based counterparts, it is still slower than local DDR memory. According to Fig. 1, we can formulate the CXL memory latency as follows:

$$t\_CXL\_Mem = t\_CXL + t\_Device\_DRAM \tag{1}$$

In the formula, t_CXL is the latency caused by the CXL stack (including the CXL packets processing, data transmission, etc.). t_Device_DRAM denotes the latency of device-side DRAMs. Although CXL-enabled CPU [33] and memory expanders [23,42] have not been commercially available till now, it has been confirmed that t_CXL_Mem is close to the latency of one-hop NUMA access

[1] We say the CXL latency is "effectively tolerated" if the performance gap between CXL and local memory is within 5%.

(i.e., `CPU-0` accessing `CPU-1`'s main memory in a dual-socket system) [28,30]. Therefore, ***t_CXL is estimated to be 50–100ns.*** [30].

To tackle CXL memory's long latency issue, some recent works focus on system-level optimizations [28,30,38]. Their main idea is to keep "hot" data in local memory while placing "cold" data in the CXL memory. Such a data mapping/migration can happen in VM instance [28] or memory page [30,38] granularity. However, these methods require complex modifications to OS kernels [30] or applications [38]. What's worse, the coarse-grained swapping methods will incur read/write amplification problems [15] and cannot fully leverage the byte-addressable and fine-grained-access advantages of CXL memory. In brief, ***there still lacks an efficient approach to reduce the CXL memory access latency directly.***

2.3 Cache Prefetching to the Rescue?

As mentioned before, a key advantage of CXL memory is the compatibility with normal CPU `load/store` interfaces, which transfer data in a cacheline granularity. Therefore, it is natural to wonder whether cache prefetching, a primary method to tolerate data access latency in conventional memory systems, can also help offset the side effects of CXL memory.

According to previous profiling [28], commercial CPU with hardware prefetchers enabled fails to tolerate the CXL latency on many tasks effectively. Given that commercial CPUs tend to equip simple and conservative hardware prefetchers [44], we also turn our eyes to some complex yet powerful prefetchers. For instance, the recently-proposed Pythia [10] prefetcher adopts Reinforcement-Learning to obtain the best prefetch policy from multiple program features and system-level feedback information. It claims to achieve the highest *coverage* and *accuracy* among CPU prefetchers. Without loss of generality, we inspect Pythia's performance under CXL memory scenarios and mainly answer two questions:

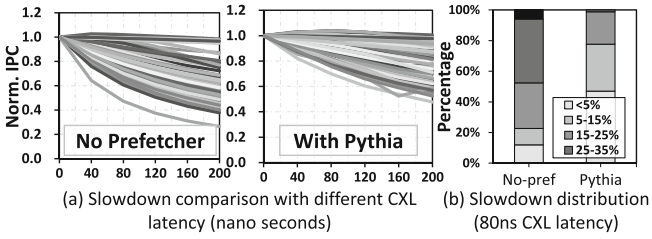

Fig. 3. Pythia's Performance on the CXL Memory.

(1) Can Powerful CPU Prefetchers Help? To answer this question, we evaluate Pythia on various SPEC2006 and SPEC2017 tasks. The simulation configurations are detailed in Sect. 4.1. As shown in Fig. 3-(a), we set t_CXL

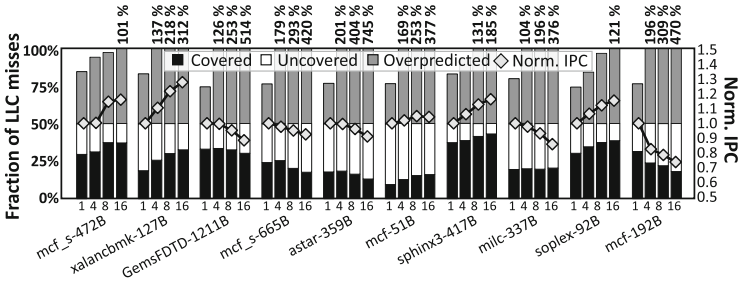

Fig. 4. Pythia's Performance with Different Prefetch Degrees.

from $0ns$ to $200ns$ and evaluate the caused slowdown. Compared to the no-prefetcher baseline, Pythia achieves more gentle slowdown curves, indicating that it can, to some extent, help tolerate the CXL latency. We also plot the slowdown distribution in Fig. 3-(b). Under a typical 80ns of CXL latency, 6% of tasks have >35% slowdown without a prefetcher, and 42% get a 25% to 35% slowdown. The fraction of unaffected tasks (slowdown <5%) is merely 12%. With Pythia, the slowdown caused by CXL latency is obviously mitigated. The fraction of unaffected tasks increases to 47% (+35%), and no task has a higher than 35% slowdown. However, 53% of cases are still heavily affected even with Pythia: 31% of tasks get a 5%–15% slowdown, and still, 22% of tasks bear a 15%–35% slowdown. ***Even powerful cache prefetchers like Pythia still leave a huge room for improvement in tolerating CXL latency.***

(2) The Impact of Prefetch Aggressiveness? In general, we can increase the aggressiveness of prefetchers (i.e., the prefetch degree) for potentially higher prefetch coverage. Therefore, we set Pythia's prefetch degree from 1 to 16 and evaluate the IPC performance, coverage, and over-prediction (i.e., the fraction of useless prefetches). As Fig. 4 shows, increasing the aggressiveness boosts the performance on half of the tasks, thanks to the improved coverage (denoted by black bars). However, on the other half, IPC gets lower with higher aggressiveness. On four of the negative cases, the coverage decreases with higher degrees. This indicates that the over-prefetched cachelines (the grey bars) evict useful cachelines, resulting in unbearable cache pollution problems. We also find that for `milc-337B`, the coverage does not change obviously, but the IPC still drops. This is due to the over-prefetched cachelines causing severe DRAM bandwidth waste. To confirm this, we also plot the bandwidth utilization in Fig. 5. For the `milc-337B` task, higher prefetch degrees result in much heavier DRAM bandwidth utilization, reflected by the increased fraction of black bars in the figure.

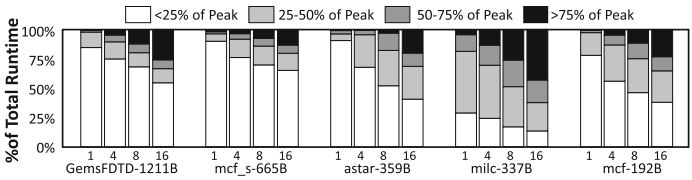

Fig. 5. Bandwidth Utilization with Different Prefetch Degrees.

Conclusions: According to these analyses we demonstrate that the prefetch coverage is the main affecting factor. Even the state-of-the-art CPU cache prefetcher, Pythia, can only help 35% of tasks tolerate the CXL latency. For the remaining tasks, it is difficult to improve the prefetch coverage further due to severe ***cache pollution*** and ***bandwidth waste*** issues. Note that these are general problems faced by CPU prefetchers since Pythia already has the (almost) highest prefetch accuracy [9,10]. Moreover, although one may want to propose better CPU prefetchers tuned for CXL memory, putting them into the host CPU will incur considerable ***CPU modification overheads.*** In a word, it is less feasible to effectively mitigate the performance gap between CXL and local DDR memory purely relying on CPU-side prefetchers.

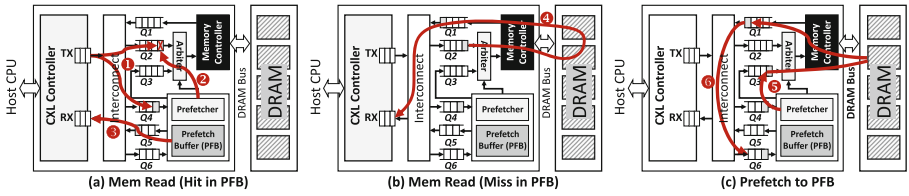

Fig. 6. Architecture and Data Path Overview of Polaris-Base.

3 Polaris

In this section, we propose to tackle the challenges mentioned above via ***memory-side prefetching***. We introduce our designed ***prefetchable*** CXL memory architectures, including Polaris-Base and Polaris-Active.

3.1 Polaris-Base Architecture

We first introduce the base design of Polaris. Figure 6 illustrates the architecture and data paths of Polaris-Base. Compared to standard CXL memory, we add a *Prefetcher* and a *Prefetch Buffer* (PFB) in the device-side controller chip. The prefetcher feeds into memory read requests decoded by the CXL controller,

performs data prefetching, and stores prefetched cachelines in PFB. As Fig. 6-(a) shows, decoded memory addresses are fed into both *Q2* (normal read queue) and *Q4* (PFB read queue) simultaneously, namely Path ❶. If a cacheline address hits in PFB while the same request is still waiting in *Q2*, it will be removed from *Q2* (Path ❷) to save DRAM bandwidth. The hit cacheline is read out from PFB via *Q5* (PFB return queue) and sent back to the CXL controller for packetization (Path ❸). If a request hits in PFB but its fork request has already been issued to the memory controller (no longer in *Q2*), the CXL controller will receive the same cacheline twice, one from PFB and the other from DRAM. It directly drops the latter one. Such a parallel-querying design removes PFB from the critical path of DRAM accesses.

If the CPU read request misses in PFB, device memory returns the missed cachelines as usual. Such a PFB-miss case is illustrated by data path ❹ in Fig. 6-(b): the memory access requests are served by the memory controller, and the read data is fed back to the CXL controller via *Q1* (DRAM return queue). Here we omit operation ❶ in this sub-figure for clarity. Received read requests are analyzed by the memory-side prefetcher. As illustrated in Fig. 6-(c), the prefetcher fetches the read addresses deposited in *Q4*, analyzes them, and issues prefetch requests to the memory controller. As path ❺ denotes, the cacheline addresses to prefetch (the prefetcher should guarantee the addresses are valid) are put into *Q3* (prefetch queue). An arbiter schedules the requests from *Q2* and *Q3* to guarantee that normal memory read has a higher priority. The prefetched data will be stored in PFB via the PFB-fill queue, *Q6* (data path ❻). Note that these queues are logically separated to explain ideas better. Some of them can be merged in physical implementation. CPU writes are not illustrated in the figure. The only thing to notice is that upon receiving a memory-write request, Polaris updates the cacheline in both PFB (if hits) and DRAM to keep consistent.

We claim that such a Polaris-Base architecture leveraging memory-side prefetching brings four main advantages:

(1) Non-intrusive Modifications: Polaris-Base restricts all modifications to the CXL memory expander and avoids costly substrate systems (e.g., the host CPU, CXL interface, OS kernels [30], system software [28], memory allocation libraries [38], etc.) modifications.

(2) Avoid CPU Cache Pollution: Polaris-Base prefetches data to the dedicated PFB buffer to create a ***Shortcut*** for future CPU accesses. It avoids polluting the host CPU cache even when an aggressive prefetcher is equipped.

(3) Harvest Device-side Memory Bandwidth: As Fig. 1 shows, the CXL memory bandwidth exposed to the hosts is jointly determined by the CXL channel bandwidth and the device-side memory bandwidth. Specifically, a standard PCIe 5.0 x16 channel provides, at most, 64 GB/s [46] of unidirectional bandwidth. However, a typical two-channel DDR5-4800 memory can provide up to 76.8 GB/s peak bandwidth, already 20% higher than the x16 channel. Polaris can harvest such over-provisioned DRAM bandwidth to facilitate memory-side prefetching.

(4) Support Complex Prefetchers: Unlike CPU prefetchers, memory-side prefetchers in standalone chips have more area/power budgets to adopt complicated prefetching mechanisms, e.g., ensembling hybrid prefetchers to improve the prefetch accuracy. We detail this idea in the following subsection.

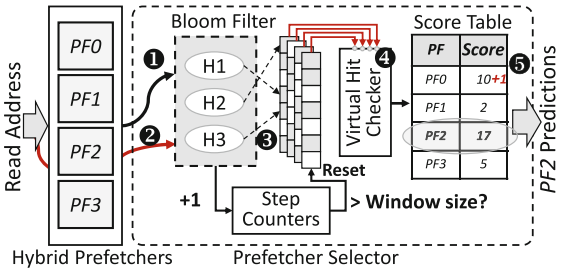

Fig. 7. Ensembled Memory-Side Prefetchers with Score-based Selector.

3.2 Ensembled Memory-Side Prefetchers

Polaris's main goal is to redirect as many memory requests as possible to the fast path (namely, improve the coverage of memory-side prefetchers) so as to reduce the average latency. However, improving the coverage of memory-side prefetchers is not an easy job. Unlike some CPU-side prefetchers, memory-side prefetchers cannot see some useful core-side information such as PC (Program Counter) [7,11,12] and branch instruction [10], etc.. Moreover, after being filtered by CPU's cache hierarchy, the memory-access patterns exposed to CXL memory become highly irregular and are harder to predict. Fortunately, Polaris equipping standalone controller chip has more resource budget for complex prefetchers. Therefore, we propose *ensemble hybrid prefetchers in* Polaris *and use a score-based selector to choose the best-performing prefetcher dynamically.* Compared to individual prefetchers, our method shows much better coverage and accuracy. Here we introduce four existing prefetchers purely adopting physical addresses as inputs that can be ensembled in Polaris:

BOP: Offset prefetching prefetches `X+D` where `X` is a line of requested address and `D` is the prefetch offset. Best-Offset prefetcher (BOP) [31] adopts a simple learning mechanism to help select the best offsets.
Domino is a temporal prefetcher [6] that records the correlations of memory accesses and prefetches correlated addresses on a trigger event (i.e., one or two cache misses).
SPP compresses the history of memory accesses to create a page signature [25]. It then correlates the signature with future likely delta patterns to make the prediction.
VLDP [41] also relies on recorded memory access history to predict future memory requests. It makes predictions based on multiple previous deltas (i.e.,the difference between two successive miss addresses in a physical page).

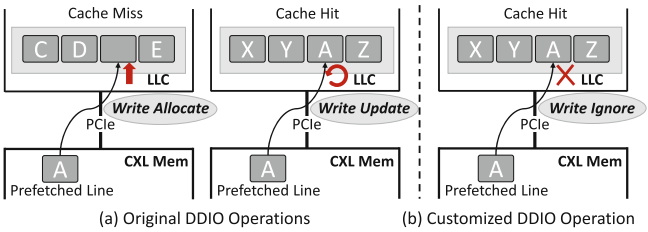

Fig. 8. Avoiding Data Overwrite with a Write-Ignore Operation.

We will demonstrate in Sect. 4.5 that these prefetchers have different advantages, and ***no prefetcher performs consistently better than the others on every task***. As shown in Fig. 7, to select the best-performing prefetcher dynamically, we design a specialized prefetcher selector based on the ***Virtual Prefetching*** mechanism [31,36]. To be specific, When receiving a memory read address, all the prefetchers (four prefetchers, *PF0* to *PF3*) generate the prefetch candidates according to their diverse prefetching mechanisms. However, these candidates ***will not actually trigger a prefetching***. Instead, they are sent to a *Bloom filter* [13] (operation ❶). Bloom filter is a low overhead probabilistic data structure used to examine whether an element is ***not*** a member of a set. The hash functions of the Bloom filter map each prefetcher's predictions to multiple entries of the corresponding *Bit vector* (operation ❸). These target entries are then set to 1. The CPU read address is mapped to certain entries of all the bit vectors to check whether this address could have been prefetched (operation ❷). For example, if all the three mapped entries in bit vector 0 have been set to 1, we assume prefetcher-0 (PF0) has prefetched the address before (***Virtual Hit***). Otherwise, if any entry's value is still zero, it means PF0 has not prefetched the address. This job is done by a virtual hit checker (❹). There is also a *Score Table* recording the gained score of each prefetcher (❺). A virtual hit increments the prefetcher's score by one each time. We always adopt the prefetcher with the highest score (e.g., *PF2* in the figure) to output the actual prefetching addresses.

The bit vectors should be reset at the beginning of each ***Step Window***: We use a per-predictor ***Step Counter*** to record the number of predictions fed into the bloom filter. The counter is reset to zero once reaching a predefined ***Window Size***) and then a new window begins. The implications are that inserting predictions (we call each bloom filter insertion a ***Step***) will gradually saturate the filter. To maintain accuracy, we have to reset the bit vectors periodically. Also, we right-shit all the scores if a score reaches the maximum number. All the components work in a pipelined manner to achieve high throughput.

3.3 Polaris-Active Architecture

Polaris-Base effectively mitigates the performance gap to local memory if the PFB-hit ratio is high enough. However, we still wonder whether we can ***make***

the most of* Polaris*'s memory-side prefetching ability to boost the system performance further. Therefore, we also propose a Polaris-Active architecture. It is featured by an *Active Prefetching* mechanism, which pushes prefetched cachelines to CPU's LLC to hide the CXL memory access latency entirely. To this end, we should answer two critical questions:

How to Push Cachelines to LLC? The mechanisms of pushing data from PCIe (CXL) devices to the CPU cache are usually referred to as Direct-Cache-Access (DCA) techniques [22,26,27,43]. For instance, Intel's DDIO (Data-Direct I/O) [24] enables a PCIe-connected device to push data into CPU's LLC cache directly. It is important to note that DDIO uses *Write-Allocate* and ***Write-Update*** policies. When a DDIO-write hits, it views the device's data as the newest and will overwrite the LLC's data (see Fig. 8-(a)). However, in our scenarios, the data in CXL memory can be older than CPU's, if CPU's dirty cachelines have not been written back. ***Directly using DDIO for active prefetching will cause severe data coherence issues.***

We argue that we can add a ***Write-Ignore*** operation to the standard DDIO protocol to solve this problem. As shown in Fig. 8-(b), if the direct-cache access request is issued by the CXL memory and the prefetched cacheline hits in CPU's LLC, the CPU just ignores the request. To support Write-Ignore, the CPU only needs to modify its DDIO control logic slightly and add a flag bit in the DDIO packets to distinguish active prefetching from normal DDIO requests.

What to Push to LLC? Considering that active prefetching consumes both the LLC's DDIO ways and the CXL channel bandwidth, it is costly to push all the prefetched data to LLC. To make better use of active prefetching, ***we only push the data with high confidence to CPU's LLC.*** Specifically, we reuse the scores (see Fig. 7) to estimate a prefetch accuracy (*Acc*):

$$Acc = \frac{Score - Score_{i-1}}{Steps\ in\ the\ i_{th}\ Window} \tag{2}$$

In this formula, *Score* denotes the running score of the best prefetcher and $Score_{i-1}$ is the old score value in the previous *Step Window*. Similar to the ensembled prefetching mechanism in Sect. 3.2, we measure the accuracy in each step window to guarantee timeliness. The prefetching accuracy is estimated by calculating the fraction of virtual prefetch hit in the current step window. When $Acc > T$, where T is a predefined threshold, we assume the prefetcher has good enough accuracy and push the cachelines to LLC via DDIO, otherwise we still store them in PFB. In practice, we can set threshold T as a power-of-two decimal like $T = 2^{-t}$. Then the controller only needs to calculate $\triangle Score = Score - Score_{i-1}$ and compare it with a $T' = \#Steps >> t$ in each step. In this way, the multi-cycle division operation is avoided. Note that the *Acc* calculation skips the first few steps (128 by default) in each window to guarantee stability. We also set an *Active Degree* parameter to limit the maximum number of cachelines that can be pushed to LLC in each prediction.

4 Evaluation

4.1 Methodology

Table 2. Default System Parameters

Core	4 GHz, 4-wide OoO, 256-entry ROB, 72/56-entry LQ/SQ
Branch Pred.	Perceptron-based, 20-cycle misprediction penalty
L1/L2 Caches	Private, 32KB/256KB, 64B line, 8 way, LRU, 16/32 MSHRs, 4 cycle/14-cycle round-trip latency
LLC	2MB/core, 64B line, 16 way, SHiP replacement, 64 MSHRs per bank, 20-cycle latency
PFB	4MB, 16 way, LRU, 20-cycle latency
CXL memory	**CXL Channel:** PCIe 5.0 x16, $t\_CXL = 80ns$ (round-trip) **DRAM:** DDR5-4800, 1 Channel, tRP, tRCD, tCAS = 16ns
CPU Prefetcher	Streamer, BOP, Pythia. Degree = 4
CXL Prefetcher	BOP, Domino, SPP, VLDP (Ensembled). Degree = 10

We compare Polaris-equipped systems against several baselines using the cycle-accurate ChampSim simulator [16]. More specifically, we adopt a modified version [2] as the code base. We customize the simulator to simulate the behavior of CXL channels and enable arbitrary CXL latency injection. We also implement the prefetch buffer (PFB) and memory-side prefetcher in the simulator.

Table 2 lists the host CPU, CXL channel, and memory configurations. We simulate a 4 GHz CPU with 1,4,8 core. Each core has a 32KB L1 cache, 256KB L2 cache, and 2 MB shared LLC. The default PFB size is 4 MB and has a 20-cycle latency. For the CXL memory, we assume the expander is based on the PCIe-5.0 x16 physical channel and has 80 ns of CXL latency. The device DRAM is a single-channel DDR5-4800 memory by default. We set $t\_CXL = 0$ and disable the PFB when simulating a local memory. The host CPU can equip one of the three CPU-side prefetchers and adopt four prefetchers to compose the ensembled memory-side prefetcher.

CPU Prefetchers: We assume the host CPU equips one of the following prefetchers: The Streamer prefetcher used by commercial CPUs [44], the Best-Offset Prefetcher (BOP) used in open-sourced RSIC-V CPU [35], and the state-of-the-art Pythia [10] prefetcher adopting reinforcement-learning techniques. The Streamer, BOP, and Pythia prefetchers are trained on L1-cache misses and fill prefetched lines into L2 and LLC. For the single-core system, the default CPU prefetching degree is set to four to achieve high coverage.

Memory-Side Prefetchers: POLARIS ensembles four hardware prefetchers introduced in Sect. 3.2, which only rely on physical addresses for prediction: BOP [31], Domino [6], SPP [25] and VLDP [41]. For the score-based prefetcher selector, we set a 512B binary vector (used by the bloom filter) per prefetcher. The window size is set to 4096, and the active prefetching threshold T is empirically set to 2^{-5}. The *Active Degree* is set to 4 by default. The detailed configurations of these hardware prefetchers are summarized in Table 3.

Table 3. Benchmarking Prefetchers

Prefetchers	Configuration	Overhead
Streamer [44]	64 trackers	0.5KB
BOP [31]	256 entry RR, MR=100, MaxScore=31, BadScore=1	1.3KB
SPP [25]	256-entry ST, 2K-entry PT, 1024-entry PF, 8-entry GHR	6.2KB
Domino [6]	128B LogMiss, 2KB Prefetch Buffer, 256B PointBuf, 64B FetchBuf.	2.4KB
Pythia [10]	2 Features, 2 Vaults, 3 Plances, 16 Actions	25.5KB

4.2 Workloads

We adopt 91 instruction traces collected from 33 workloads of SPEC2006 [21], SPEC2017 [14], PARSEC-2.1 [1] and GAPBS [8] benchmarks for evaluation. They are summarized in Table 4. These traces, except for GAPBS, are obtained from Pythia's repo [2]. We record GAPBAS traces manually using Champsim's tracer with a `[-u 20]` running arguments. For GAPBS, we use 150M instructions for warmup and 50M for evaluation. The other traces use 100M instructions for warmup and 100M for evaluation. All traces have higher than 3 MPKI running on a no-prefetcher system.

Table 4. Workloads for evaluation

Benchmark	#Workloads	#Traces	Example Workloads
SPEC2006	13	38	gcc,mcf,lbm,libquantum,
SPEC2017	10	35	gcc,mcf,pop2,fotonik3d,
PARSEC	4	12	canneal,facesim,fluidanimate,
GAPBS	6	6	bfs,pagerank,spmv,bc

4.3 Performance Metric

We first define a ***Slowdown*** function as the performance metric to compare among different system configurations:

$$Slowdown(\Omega, \Pi) = \frac{IPC(\Omega, \Pi)_{CXL} - IPC(\Omega)_{Local}}{IPC(\Omega)_{Local}} \tag{3}$$

In this formula, Ω and Π represent the adopted CPU-side and memory-side prefetching mechanisms, respectively. Specifically, $\Omega \in$*{None, Streamer, BOP, Pythia}* and $\Pi \in \{Polaris - Base, Polaris - Active\}$. Our primary goal is to make the system's IPC on CXL memory, namely $IPC(\Omega, \Pi)_{CXL}$, close to or higher than the baseline system's, which adopts the same CPU-side prefetcher but using local DDR memory, namely $IPC(\Omega)_{Local}$. Ideally, the slowdown should be close to or even higher than zero to indicate that the performance gap between CXL and local memory is effectively mitigated.

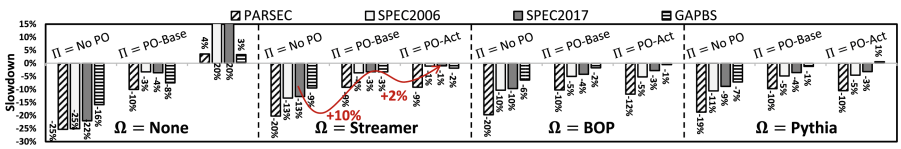

Fig. 9. Slowdown Mitigation with Polaris

4.4 Performance Overview

Performance with Single Task: We first evaluate Polaris's performance on the single-core system, which runs a single task each time. We compare the average slowdown under various (Ω, Π) configurations in Fig. 9. In the figure, `No PO` denotes the baseline system with no memory-side prefetchers, `PO-Base` and `PO-Act` are short for Polaris-Base and Polaris-Active architectures. We can observe that with Polaris, the average slowdown on all four benchmarks is substantially mitigated. Without memory-side prefetching (Π =`NO PO`), the system bears –6% (Ω = `BOP`, GAPBS) to –25% (Ω=`None`, PARSEC and SPEC2006) average slowdown. With Polaris-Base, the average slowdown is only –1% to –10%. Polaris-Active mitigates the slowdown further on many cases. For instance, as annotated by the red line, with Ω = `Streamer`, Polaris-Base has already reduced the slowdown on SPEC2017 (the dark bars) by 10%. Polaris-Active reduces the value by 2% further. Surprisingly, Polaris-Active even achieves higher IPC than the local-memory system without CPU-side prefetchers (Ω =`None`). This is because Polaris-Active can directly push cachelines to CPU's LLC, compensating for the absence of a CPU-side prefetcher. In rare cases, Polaris-Active performs slightly worse than Polaris-Base (Ω = `BOP`, PARSEC). This may be because some useful cachelines are evicted by prefetched

ones, even with the DDIO capacity constraints. Fortunately, the negative case still outperforms the `NO PO` baseline by eight points.

Figure 10 also presents the breakdown of the slowdown on all traces. Polaris-Base and Polaris-Active can increase the percentage of unaffected tasks (slowdown <5%) by 26% (Pythia) to 85% (No Pref.), 43% on average. They can also substantially mitigate the ratio of heavily-affected tasks denoted by the dark bars. For example, Polaris-Act saves 43 out of 44 tasks from suffering >25% slowdown in the No Pref. (Ω = `None`) system. The ratio ranges from 70% to 98% with different CPU prefetchers.

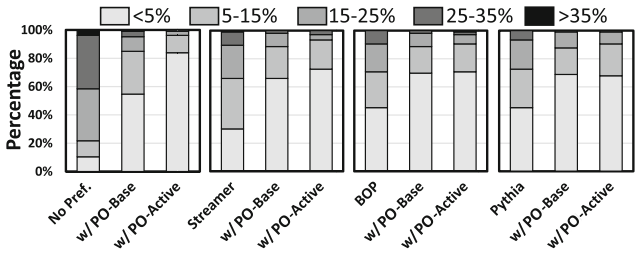

Fig. 10. Breakdown of Slowdown on All Tasks.

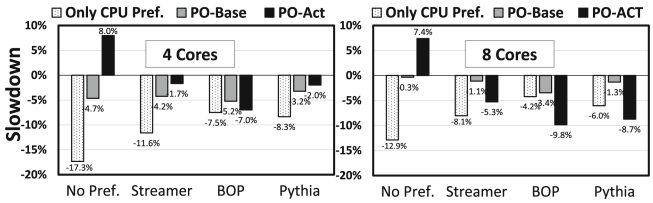

Fig. 11. Performance with Multi-Tasks.

Performance with Multi-tasks: We then evaluate Polaris's performance on multi-core systems, with each core running a different task. We increase the number of cores to 4 and 8 and set two DRAM channels to match the bandwidth requirements. For an N-core system, we randomly select N traces from the 91 traces to build a mixed trace. We prepare eight mixed traces for each configuration and calculate the geomean IPC of all the cores as the multi-core IPC. The CPU prefetch degrees are reduced from four to two in the multi-core systems. As shown in Fig. 11, in the four-core system, Polaris-Base mitigates a 2.3% to 12.6% slowdown when cooperating with different CPU-side prefetchers.

Polaris-Active gets a higher IPC than the local-memory baseline by +8.0%. With Streamer or Pythia as the CPU prefetcher, Polaris-Active pushes the

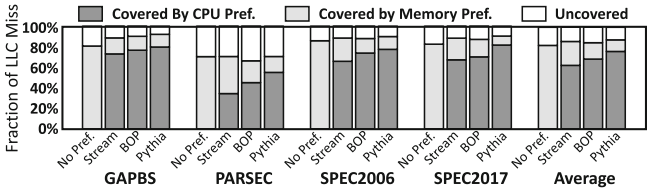

Fig. 12. Coverage Improvement with Polaris-Base.

slowdown to a much lower value than Polaris-Base, merely −1.7% and −2.0%, respectively. However, Polaris-Active is less effective than Polaris-Base on a BOP-equipped system. We infer that this is because BOP generates too many miss-predicted prefetch requests, based on which Polaris-Active can hardly ensure high active-prefetching accuracy, either. Such a phenomenon is more severe in the eight-core system. As we can see in the right figure, Polaris-Active works perfectly without a CPU-side prefetcher, but works more poorly than Polaris-Base and even hurts the performance in the BOP and Pythia-based systems. We infer that with more working threads, the DDIO ways and CXL bandwidth are stressed greatly. More conservative active prefetching parameters (i.e., higher threshold T and lower *Active Degree*, etc.) may be beneficial. We leave the study of the optimal parameters setting to our future work.

4.5 Performance Analysis

Coverage Improvement: To better interpret Polaris's effectiveness, we profile the prefetch coverage in the baseline systems equipping Polaris-Base. As shown in Fig. 12, we break down total LLC misses into three parts: 1) Covered by CPU prefetcher. 2) Covered by Polaris's prefetcher and 3) Uncovered LLC misses. Firstly, we can easily observe that, when $\Omega = \{\texttt{Streamer,BOP,Pythia}\}$ the CPU-side prefetchers can cover 35% to 80% LLC misses. Based on CPU prefetchers, Polaris can further reduce 34% to 66% of uncovered LLC misses, 54% on average. We also find that, when the host CPU does not equip a prefetcher, about 70% to 85% of LLC misses are hit in the PFB.

Score-Based Ensembled Prefetchers: We compare the score-based ensembled prefetcher (see Sect. 3.2) with every individual prefetcher. We adopt the representative SPEC traces used in Fig. 4 for demonstration. As shown in Fig. 13, no individual prefetcher performs consistently better than the others among all tasks (Red circles annotate the best-performing tasks of each prefetcher). We also find that the proposed ensembled prefetcher (the black bars) can achieve near-optimal speedup on almost all tasks.

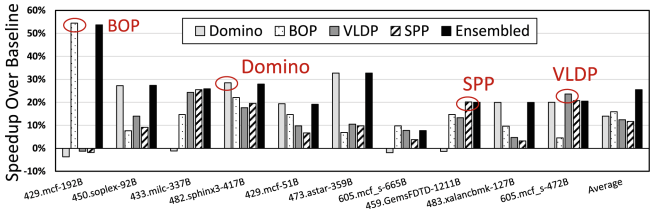

Fig. 13. Benefits of the Ensembled Memory-side Prefetcher.

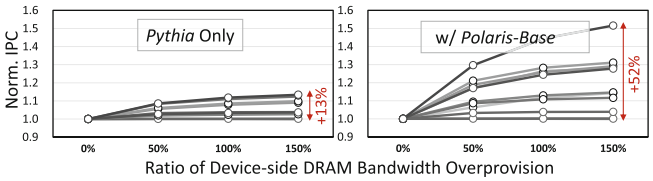

Fig. 14. Speedup Comparison with Different Over-provision Ratio η.

4.6 Sensitivity Analysis

DRAM Bandwidth Over-provision: As claimed before, an advantage of Polaris is the ability to harvest the higher device-side DRAM bandwidth for prefetching. We use the over-provision ratio $\eta = \frac{DRAM\_Bandwidth}{CXL\_Bandwidth} - 1$ to measure how much device-side DRAM bandwidth is over-provisioned. Without loss of generality, we compare the performance of a Pythia + Polaris-Base and a Pythia-only system under different η values. Following Pythia's practice [10], we constrain the single-core system's CXL bandwidth to 8 GB/s and set the default DRAM IO speed to 1000MTPS ($\eta = 1$) to emulate the bandwidth budget in multi-core systems. We test on PARSEC tasks since they have the worst performance among all four benchmarks. As compared in Fig. 14, for the baseline system without Polaris, over-providing 150% device-side DRAM bandwidth only brings 13% IPC improvement. With Polaris, the system's performance improves by up to 52% with higher device-side DRAM bandwidth. This indicates that Polaris effectively leverages the over-provided DRAM bandwidth to facilitate memory-side prefetching.

PFB Size: We set different PFB sizes ranging from 512KB to 8 MB and use the SPEC traces for a quick exploration on the Polaris-Base system. The results are shown in Fig. 15. It is interesting to find that an accurate CPU-side prefetcher, namely Pythia is more sensitive to the PFB size. An 8MB PFB brings about a 14% performance improvement over the 512KB PFB. While for Ω =`None` or `Streamer`, the IPC increases slowly. We infer that this is because Polaris does not distinguish between demand cache misses and CPU prefetch misses. If the CPU prefetcher's predictions are accurate, Polaris's prefetcher is more likely to generate useful prefetch-on-prefetch requests, which demand a larger PFB to

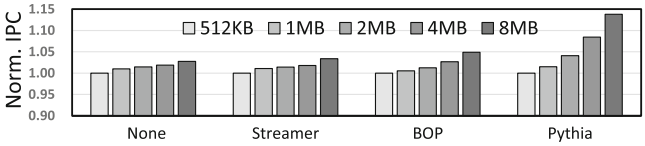

Fig. 15. Polaris-Base's Performance with Different PFB Sizes.

store. Otherwise, Polaris may generate too many inaccurate prefetches, which does not easily benefit from a larger prefetch buffer.

4.7 Overhead of Polaris

Similar to previous works [6,25,31,36,39], we assume the main overhead of Polaris's prefetcher comes from the storage. As listed in Table 3, the ensembled prefetchers consume roughly 35.9KB of SRAM. Taking into consideration the bit vectors and score tables, etc., we assume a 40KB budget. We estimate the power and area using Synopsys Design Compiler 2016 with FreePDK 45 nm library [34]. The registers have 2.82 mm^2 total cell areas and consume about 240.8 mW of power. We also estimate the overhead of the 4MB PFB via CACTI [32] under the 40 nm technology. The 16-way PFB consumes 24.28 mm^2 of area and 1.53 W of peak power. Putting them together, Polaris roughly requires 27.1 mm^2 more area and a 1.77 W additional power budget.

5 Conclusion

This paper presents Polaris, a novel CXL memory featured by memory-side prefetching. It enhances the system's prefetching capability while avoiding CPU cache pollution and mitigating bandwidth waste. Polaris's base design does not incur substrate-system modifications to be drop-in compatible with data center servers. If one permits small CPU changes, Polaris can actively push prefetched cachelines to CPU's LLC to boost performance further. Polaris is the first attempt to bring some conventional CPU-side tasks, like cache prefetching, to the CXL-device side for more opportunities.

Acknowledgment. This work is supported by Key-Area Research and Development Program of Guangdong Province (2021B0101310002), NSFC (61832020, 62032001, 92064006) and 111 Project (B18001).

References

1. Parsec 2.1, 2022.9. https://parsec.cs.princeton.edu/
2. Pythia's github repo, 2022.9. https://github.com/CMU-SAFARI/Pythia
3. Aguilera, M.K., et al.: Remote regions: a simple abstraction for remote memory. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18), pp. 775–787 (2018)

4. Al Maruf, H., Chowdhury, M.: Effectively prefetching remote memory with leap. In: 2020 USENIX Annual Technical Conference (USENIX ATC 20), pp. 843–857 (2020)
5. Amaro, E., et al.: Can far memory improve job throughput? In: Proceedings of the Fifteenth European Conference on Computer Systems, pp. 1–16 (2020)
6. Bakhshalipour, M., Lotfi-Kamran, P., Sarbazi-Azad, H.: Domino temporal data prefetcher. In: 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 131–142. IEEE (2018)
7. Bakhshalipour, M., Shakerinava, M., Lotfi-Kamran, P., Sarbazi-Azad, H.: Bingo spatial data prefetcher. In: 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 399–411. IEEE (2019)
8. Beamer, S., Asanović, K., Patterson, D.: The gap benchmark suite, arXiv preprint arXiv:1508.03619 (2015)
9. Bera, R., et al.: Hermes: accelerating long-latency load requests via perceptron-based off-chip load prediction. In: 55th IEEE/ACM International Symposium on Microarchitecture, MICRO 2022, Chicago, IL, USA, 1–5 October 2022. IEEE, 2022, pp. 1–18 (2022). https://doi.org/10.1109/MICRO56248.2022.00015
10. Bera, R., Kanellopoulos, K., Nori, A., Shahroodi, T., Subramoney, S., Mutlu, O.: Pythia: a customizable hardware prefetching framework using online reinforcement learning. In: MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture, 2021, pp. 1121–1137 (2021)
11. Bera, R., Nori, A.V., Mutlu, O., Subramoney, S.: Dspatch: dual spatial pattern prefetcher. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019, pp. 531–544 (2019)
12. Bhatia, E., Chacon, G., Pugsley, S., Teran, E., Gratz, P.V., Jiménez, D.A.: Perceptron-based prefetch filtering. In: 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA), pp. 1–13. IEEE (2019)
13. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970). https://doi.org/10.1145/362686.362692
14. Bucek, J., Lange, K.-D., Kistowski, J.V.: SPEC CPU2017: next-generation compute benchmark. In: Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, pp. 41–42 (2018)
15. Calciu, I., et al.: Rethinking software runtimes for disaggregated memory. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 79–92 (2021)
16. ChampSim, Champsim simulator, 2022.9. https://github.com/ChampSim/ChampSim
17. C. foundation, Cxl 3.0 specification, 2022.9. https://www.computeexpresslink.org/download-the-specification
18. Gao, Y., et al.: When cloud storage meets $\{RDMA\}$. In: 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21), pp. 519–533 (2021)
19. Gouk, D., Lee, S., Kwon, M., Jung, M.: Direct access,$\{High - Performance\}$ memory disaggregation with $\{DirectCXL\}$. In: 2022 USENIX Annual Technical Conference (USENIX ATC 22), pp. 287–294 (2022)
20. Gu, J., Lee, Y., Zhang, Y., Chowdhury, M., Shin, K.G.: Efficient memory disaggregation with infiniswap. In: 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), pp. 649–667 (2017)
21. Henning, J.L.: SPEC CPU2006 benchmark descriptions. ACM SIGARCH Comput. Archit. News **34**(4), 1–17 (2006)

22. Huggahalli, R., Iyer, R., Tetrick, S.: Direct cache access for high bandwidth network i/o. In: 32nd International Symposium on Computer Architecture (ISCA'05), pp. 50–59. IEEE (2005)
23. Hynix, S.: Sk hynix cxl memory expander, 2022.9. https://news.skhynix.com/sk-hynix-develops-ddr5-dram-cxltm-memory-to-expand-the-cxl-memory-ecosystem/
24. Intel, Intel data-direct io, 2022.9. https://www.intel.cn/content/www/cn/zh/io/data-direct-i-o-technology.html
25. Kim, J., Pugsley, S.H., Gratz, P.V., Reddy, A.N., Wilkerson, C., Chishti, Z.: Path confidence based lookahead prefetching. In: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 1–12. IEEE (2016)
26. Kumar, A., Huggahalli, R., Makineni, S.: Characterization of direct cache access on multi-core systems and 10gbe. In: 2009 IEEE 15th International Symposium on High Performance Computer Architecture, pp. 341–352. IEEE (2009)
27. León, E.A., Ferreira, K.B., Maccabe, A.B.: Reducing the impact of the memorywall for I/O using cache injection. In: 2007 15th Annual IEEE Symposium on High-Performance Interconnects (HOTI), pp. 143–150. IEEE (2007)
28. Li, H., et al.: First-generation memory disaggregation for cloud platforms, arXiv preprint arXiv:2203.00241 (2022)
29. Lim, K., Chang, J., Mudge, T., Ranganathan, P., Reinhardt, S.K., Wenisch, T.F.: Disaggregated memory for expansion and sharing in blade servers. ACM SIGARCH Comput. Archit. News **37**(3), 267–278 (2009)
30. Maruf, H.A., et al.: TPP: transparent page placement for cxl-enabled tiered memory, arXiv preprint arXiv:2206.02878 (2022)
31. Michaud, P.: Best-offset hardware prefetching. In: 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 469–480 (2016)
32. Muralimanohar, N., Balasubramonian, R., Jouppi, N.P.: Cacti 6.0: a tool to model large caches. HP Lab. **27**, 28 (2009)
33. Nassif, N., et al.: Sapphire rapids: the next-generation intel Xeon scalable processor. In: 2022 IEEE International Solid-State Circuits Conference (ISSCC), vol. 65, pp. 44–46. IEEE (2022)
34. NCSU, Freepdk45. https://www.eda.ncsu.edu/wiki/FreePDK45:Contents
35. OpenXiangShan, Xiangshan riscv cpu, 2022.9. https://github.com/OpenXiangShan/XiangShan
36. Pugsley, S.H., et al.: Sandbox prefetching: safe run-time evaluation of aggressive prefetchers. In: IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), pp. 626–637. IEEE (2014)
37. Ruan, Z., Schwarzkopf, M., Aguilera, M.K., Belay, A.: $\{AIFM\}$:$\{High-Performance\}$,$\{Application-Integrated\}$ far memory. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pp. 315–332 (2020)
38. Samsung, Smdk, 2022.9. https://github.com/OpenMPDK/SMDK.git
39. Shakerinava, M., Bakhshalipour, M., Lotfi-Kamran, P., Sarbazi-Azad, H.: Multi-lookahead offset prefetching. The Third Data Prefetching Championship (2019)
40. Shan, Y., Huang, Y., Chen, Y., Zhang, Y.: $\{LegoOS\}$: a disseminated, distributed $\{OS\}$ for hardware resource disaggregation. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pp. 69–87 (2018)
41. Shevgoor, M., Koladiya, S., Balasubramonian, R., Wilkerson, C., Pugsley, S.H., Chishti, Z.: Efficiently prefetching complex address patterns. In: 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 41–152. IEEE (2015)

42. Sumsung, Expanding the limits of memory bandwidth and density: Samsung's cxl dram memory expander, 2022.9. https://semiconductor.samsung.com/newsroom/tech-blog/expanding-the-limits-of-memory-bandwidth-and-density-samsungs-cxl-dram-memory-expander/
43. Tang, D., Bao, Y., Hu, W., Chen, M.: DMA cache: using on-chip storage to architecturally separate I/O data from CPU data for improving I/O performance. In: HPCA-16: The Sixteenth International Symposium on High-Performance Computer Architecture, pp. 1–12. IEEE (2010)
44. Viswanathan, V.: Disclosure of H/W prefetcher control on some intel processors. Intel SW Developer Zone (2014)
45. Wang, C., et al.: Semeru: a $\{Memory-Disaggregated\}$ managed runtime. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pp. 261–280 (2020)
46. Wiki, Pcie 5.0, 2022.9. https://en.wikipedia.org/wiki/PCI_Express